ISO Doc No: WG21/N0829
ANSI Doc No: X3J16/96-0011
Date: January 29, 1996
Reply To: Erwin Unruh
erwin.unruh@mch.sni.de

Some `operator new` Topics

Erwin Unruh

Siemens Nixdorf Informatiossysteme AG
Department of Software Development Systems
C/C++ Front-End Laboratory LoB BS2000 SD 21
Otto-Hahn-Ring 6
D–81739 München
Germany

1 Introduction

In my early days of C++ I read the ARM and realized the problem of `operator new`. I even wrote an email to one of the committee members. I did not get an answer, but the problem remain in my head (and under a thick layer of other papers).

When the topic of placement new came up, I took note and did re-organize my thaughts. Here are a few messages from the reflektor, which relate to the topic. The first gives the rationale, the second some proposlas and the third a comment from Bill Gibbons.

2 Message 6228

The current WP mixes two independent problems:

First: The language expressions `new` and `delete` should be defined to just call a few functions. There are requirements about the accessibility and scoping of these functions, but no more.

So: `new (place) T (init)` does the following: lookup `operator new` by looking in the class T (if it is a class), otherwise in the global scope. A set of magic ones is considered to be declared at file scope for purpose of this lookup. Then do overload resolution and accessibility check for these operators. Do overload resolution on the constructors. Check whether an `operator delete` is necessary for exception handling. These are all checked at compile time.

At run time, the `operator new` is called. After that the constructor is called. If the `operator new` returns *zero*, the constructor is not called (or the behaviour is implementation defined). There is no requirement about the `operator new` yielding a storage disjunct from other objects. Problems with that sort of allocation is covered by the object model.

Second: There is a set of requirements which hold for the library version of `operator new` and `operator delete`. Here it is described that a pointer given to the `(::STD) ::operator delete` must be a pointer eventually created by a call to `(::STD) ::operator new(size_t)` or `(::STD) ::operator new(size_t,nothrow)`.

There is also a requirement that allocation done by the standard library imposes some restrictions on the allocation function.

If some other `operator delete` is called (a user defined) it is up to the programmer to state its own requirements. If his `operator delete` does nothing, there would be no requirement on the pointer at all.

Third: There should be a **non-normative** section about usual behaviour of allocation functions. There may even be some normative rules about them, but any violation must be taken as undefined behaviour. A compiler is not able to detect behaviour of user-written functions.

As a conseqence, the following program would be well-formed:

```
class A {
      static A mya;
      A(const A&);
      A& operator=(const A&);
public:
      int i;

      A() : i(0) {}
      ~A() {}

      void* operator new(size_t) { return &A::mya; }
      void operator delete(size_t) {}
};
```

This class lays all heap objects on the same place. Since constructor and destructor are quite trivial and the class is a POD, it is allowed to call the constructor over the same object again and again. So allocating a bunch of A objects would just give you the same object every time.

If this class is ill-formed, show an algorithm detecting it!

3 Message 6306

When the discussion of `operator new (size_t, nothrow)` came up, I argued for a more general approach. After having a more detailed look at the WP I realized that most of the rules are already in the right place. There are however a few necessary changes and some new issues.

I enumerate the small topics I cover and give WP changes and rationale.

3.1 calling allocation with less than `sizeof(T)`

The WP does not allow the compiler to allocate more than `sizeof(T)` bytes when allocating storage for an object of type T. It does however allow to allocate less storage. For

```
struct X { inta; char b;}; new X;
```

it may call the allocation function with the value of 5 since 5 Bytes are enough to fit a X in. The `sizeof(X)` however is 8 since it must account alignment.

Issue: is this intentional? The example in paragraph 12 indicates that the allocated storage must be exactly `sizeof(T)` for objects.

WP change [expr.new] 5.3.4/9 change text in parenthesis to:

> which shall be no less than the size of the object and be greater than the size only if the object is an array.

3.2 requirements to allocation function

The current rules for allocation outlaws some usefull allocation techniques, where a **big** object is allocated from `::operator new` and little chunks are given to a much smaller class. The problems of overlapping objects are dealt with in the object model. The requirements to an allocation function are much less strict. They are:

1. return a pointer p
2. it must be suitably aligned (implementation defined)
3. the pointers p through p+n-1 must be valid pointers and must be derefencable
4. For arrays: p+n must be a valid pointer

   The other rules are either too strict or are covered by other rules:

5. disjoint allocation: see object model
6. zero will be unique: violation will create "the same object"

WP change [basic.stc.dynamic.allocation] 3.7.3.1/2 needs some work.

3.3 deleting pointers obtained with placement

The rule that deleting an object obtained via a placement new is simply wrong. The real issue is that the library delete will cover storage obtained via library new, but not library placement void* new. If a user writes his own new and delete, he may do something reasonable. The behavior of the library delete is already described in chapter 18.

WP change [expr.delete] 5.3.5/2 twice remove the words

> without a new-placement specification.

3.4 (editorial) add new(nothrow) to replacables

WP change [basic.stc.dynamic] 3.7.3 add operator new(nothrow) and its array version to the list of replaceable functions

3.5 allocation returning zero

3.7.3.1/4 says: If the allocation function returns the null pointer the result is implementation defined.

This means that any use of new(nothrow) directly depends on implementation defined behaviour. This surely is an issue and I would like the behaviour of an allocation function returning null being the new expression yielding null.

4 Message 6310

From Bill Gibbons

(In core-6306, Erwin Unruh discusses new/delete.)

I agree completely. Now that we have a memory model, the semantics of a new-expression and delete-expression should be described entirely in terms of what functions are called (various forms of operator new/delete, constructors and destructors).

The constraints on the arguments to the functions belong in the library clauses. The constraints on the use of memory belong in the memory model.

Other than describing what functions are called, the description of new-expressions and delete-expressions should have a non-normative note saying that if any user-written operator new/delete is called, it's the user's responsibility to ensure that it will obey the memory model and operator new/delete constraints. For example, if a pointer returned by a user-written operator new is eventually passed to the default operator delete, it's the user's responsibility to ensure that the call will work.